Defence Research and Development Canada
Recherche et développement pour la défense Canada

AISS Miro Manual

A Rough Guide

J. Giesbrecht, J. Collier, G. Broten, S. Verret, S. Monckton
DRDC Suffield

Technical Memorandum
DRDC Suffield TM 2006-115
December 2006

Canada

AISS Miro Manual

A Rough Guide

J. Giesbrecht, J. Collier, G. Broten, S. Verret and S. Monckton
Defence R&D Canada – Suffield

Defence R&D Canada – Suffield

Technical Memorandum

DRDC Suffield TM 2006-115

December 2006

Principal Author

Original signed by J. Giesbrecht

J. Giesbrecht

Approved by

Original signed by D.M. Hanna

D. M. Hanna

Head/AISS

Approved for release by

Original signed by Dr P.A. D'Agostino

Dr P. A. D'Agostino

Head/Document Review Panel

Abstract

The Autonomous Land Systems (ALS) and Cohort projects undertaken by the Autonomous Intelligent System Section (AISS) at Defence R&D Canada – Suffield research and develop autonomous unmanned vehicles. A software architecture provides a common software framework allowing researchers to easily implement, test, and share their research or algorithms. After a thorough review of existing robotics toolkits, the "Miro" framework was chosen as a basis on which to build DRDC's Architechture for Autonomy. This document gives the reader background on "Miro" and the detailed procedure used to install the toolkit and all associated applications. Second, the basic concepts of "Miro" are described including the Interface Definition Language, Naming Service, Polled Mode Data Transfer, Event Driven Data Transfer and the Miro Directory Structure. Third, various "Miro" utilities and several "Miro" examples are described. Finally, more detailed "Miro" examples including how to add services to the Automake structure, the OMG Interface Definition Language and Parameter Files are discussed.

Résumé

Les projets Systèmes terrestres autonomes (STA) et Cohort entrepris par la Section des systèmes intelligents autonomes (SSIA) à R & D pour la défense – Suffield effectuent la recherche et la mise au point des véhicules autonomes sans équipage. Une architecture logicielle procure un cadre conceptuel de logiciels commun permettant aux chercheurs d'implémenter les essais, de les tester facilement et de mettre leur recherche ou algorithmes en commun. Après avoir effectué une étude approfondie des trousses à outils de logiciels robotiques existants, le cadre conceptuel « Miro » a été choisi comme base sur laquelle construire Architecture pour l'autonomie de RDDC. Ce document fait l'historique de « Miro » pour le lecteur et lui donne la procédure détaillée ayant été utilisée pour installer la trousse à outils et toutes les applications qui y sont reliées. Les concepts de base de « Miro » sont décrits en second, y compris le langage de définition de l'interface, les services de dénomination, le transfert de données par appels sélectifs, le transfert de données guidé par l'événement et la structure de répertoire de Miro. Troisièmement, on y décrit une variété de logiciels utilitaires et plusieurs exemples « Miro ». Enfin, on discute d'exemples additionnels détaillés « Miro » dont comment ajouter des services à la structure Automake, au Langage de définition ayant une interface GOM et aux fichiers de paramètres.

This page intentionally left blank.

Executive summary

AISS Miro Manual

J. Giesbrecht, J. Collier, G. Broten, S. Verret, S. Monckton; DRDC Suffield TM 2006-115; Defence R&D Canada – Suffield; December 2006.

Background: The Autonomous Land Systems (ALS) and Cohort projects, undertaken by the Autonomous Intelligent System Section (AISS) at Defence R&D Canada – Suffield, research and develop autonomous unmanned vehicles. In order for this research to progress, a software architecture is required that allows researchers to develop, implement and test their research. Such an architecture provides a common software framework through which Defence Scientists can easily collaborate. This collaboration is especially important for complex undertakings such as autonomous systems. After a thorough review of existing robotics toolkits, the "Miro" framework was chosen as the framework for DRDC work.

Principal Results: The Miro framework enables software programs distributed across a network to transparently communicate with each other. It contains many powerful tools that allow software components and algorithms to be combined for autonomous robotic systems applications. Miro also contains utilities commonly required in developing robot systems, such as the logging and playback of vehicle data. Using the Miro framework an individual researcher can, with relative ease, integrate his/her contribution into overall unmanned systems. This document is a guide to DRDC scientists who use the Miro toolkit to develop algorithms for autonomous systems.

Significance of Results: DRDC used Miro on the Raptor unmanned ground vehicle to successfully demonstrate autonomous capabilities. The adaptation of Miro and the implementation of autonomous capabilities developed under the ALS project can be difficult for new users. This document aids them in using DRDC's Miro implementation by outlining installation procedures and providing examples for understanding and developing Miro based software. It also dissects existing code examples to provide a deeper understanding of the software involved. Using this document, a user should be able to quickly contribute to Miro based autonomy projects within AISS.

Future Work: Within the Miro framework, DRDC scientists have already developed a number of useful algorithms, which interact in specific ways, to create intelligent robotic behaviours. Although this manual describes, in detail, Miro's configuration and usage, it does not address issues associated with implementing autonomous capabilities. The ALS Project, with its development of the Raptor unmanned ground vehicle, used the Miro framework as a basis for implementing autonomy. A companion guide, describing the Raptor implementation, is required by researchers wishing to develop autonomous capabilities.

Sommaire

AISS Miro Manual

J. Giesbrecht, J. Collier, G. Broten, S. Verret, S. Monckton; DRDC Suffield TM 2006-115; R & D pour la défense Canada – Suffield; décembre 2006.

Contexte: Les projets Systèmes terrestres autonomes (STA) et Cohort entrepris par la Section des systèmes intelligents autonomes (SSIA) à R & D pour la défense – Suffield effectuent la recherche et la mise au point des véhicules autonomes sans équipage. Une architecture logicielle, permettant aux chercheurs de mettre au point, d'implémenter et de tester cette recherche, est requise pour que la recherche progresse. Une telle architecture procure un cadre conceptuel de logiciels commun qui permet aux chercheurs de la défense de pouvoir aisément collaborer. Cette collaboration est surtout importante pour les opérations complexes telles que les systèmes autonomes. Après avoir effectué une étude approfondie des trousses à outils de logiciels robotiques existants, le cadre conceptuel « Miro » a été choisi comme base sur laquelle construire Architecture pour l'autonomie de RDDC.

Résultats principaux: Le cadre conceptuel Miro permet aux programmes logiciels distribués sur le réseau de communiquer de manière transparente les uns avec les autres. Il contient beaucoup d'outils puissants permettant de combiner des composantes des logiciels avec des algorithmes pour les applications de systèmes robotiques autonomes. Miro contient aussi des logiciels utilitaires, généralement requis pour la mise au point de systèmes de robots, tels que l'enregistrement chronologique et la lecture des données d'un véhicule. Un chercheur individuel peut utiliser le cadre conceptuel Miro pour intégrer, avec une certaine aisance, sa contribution à un ensemble de systèmes sans équipage. Ce document est un guide destiné aux scientifiques de RDDC qui utilisent la trousse à outils Miro pour mettre au point les algorithmes des systèmes autonomes.

Portée des résultats: RDDC a utilisé Miro sur le véhicule terrestre sans équipage Raptor pour démontrer avec succès ses capacités d'autonomie. L'adaptation de Miro et l'implémentation de ses capacités autonomes mise au point par le projet STA peut être difficile pour les nouveaux utilisateurs. Ce document les aide à utiliser l'implémentation Miro de RDDC en soulignant les procédures d'installation et en procurant des exemples pour comprendre et développer des logiciels basés sur Miro. Il dissèque aussi des exemples de codes existants pour procurer une meilleure compréhension du logiciel utilisé. Un utilisateur devrait être capable de contribuer rapidement aux projets d'autonomie basés sur Miro, au sein de la SSIA, en utilisant ce document.

Travaux futurs: Les chercheurs de RDDC ont déjà mis au point, dans le cadre conceptuel Miro, un certain nombre d'algorithmes utiles qui interagissent de manière spécifique pour créer des comportements robotiques intelligents. Bien que ce manuel décrive en détail les configurations et les usages de Miro, il n'aborde pas les problèmes reliés à l'implémentation des capacités autonomes. Le projet STA a utilisé le cadre conceptuel Miro, durant la mise au point du véhicule terrestre sans équipage, comme élément de base pour implémenter l'autonomie. Un guide complémentaire, décrivant l'implémentation du Raptor, est nécessaire pour les chercheurs souhaitant développer des capacités autonomes.

Table of contents

List of figures

1 Introduction

Miro is MIddleware for RObots and is a distributed component based framework for mobile robot control based on CORBA. Put in simple terms, Miro is a *framework* that simplifies the process of building a robot by providing capabilities that are commonly used by robot systems.

The ALS program and the Cohort ARP[1] researches and develops autonomous unmanned vehicles. In order for this research to occur and progress, an *architecture* is required that allows the researcher to implement and test their research or algorithms. The AISS section thoroughly reviewed the requirements for autonomous intelligent systems in a Technical Report [1] which concluded:

> "To effectively distribute intelligence modules within and between UVs a layered modular hardware design and portable, maintainable coding practice require an architecture that, at once, inherently supports and encourages distributed computing, and frees investigators to focus on the development of intelligent single and multi-vehicle control systems. An architecture founded on these elements defines, at a high level, the links between various software components that create an operational vehicle. Ideally, architectures should seamlessly transition between real vehicle control, system diagnosis through the replay of gathered data and the control of a vehicle in a simulated world. Ideally, the investigator is then free to develop intelligence algorithms without vehicle implementation distractions. When satisfied with the simulated performance, the investigator can safely execute algorithms on a physical vehicle. Conversely, with the vehicle operating, data from the environment can be gathered, archived, and replayed within a simulated environment to investigate, debug and optimize the performance of an algorithm."

This technical report investigated *architectures for autonomy* that would meet the current and future requirements of autonomous intelligent systems and concluded:

> "... the Miro framework and ACE/TAO foundations exemplify highly modular, extensible, and reusable components that offer direct research benefits. Extensible, scalable, distributed, and modular components will ease installation across DRDC systems and, significantly, will simplify cooperation with other research institutions. Components achieve these goals by using defined interfaces to share information between processes. These critical interfaces permit sharing of information with other components, inherently allowing information distribution. The distribution details remain hidden from the researcher, allowing full research effort to focus on increasing vehicle capabilities."

Through the use of Miro an individual researcher can, with relative ease, integrate his/her contribution to the overall unmanned system (UxV) whether that contribution be sensing,

[1] And future autonomous intelligent systems Advanced Research Programs (ARPs).

mapping, navigation, localization, arbitration, mobility, etc. In summary, all researchers of the Autonomous Intelligent Systems Section (AISS) **must** be familiar with Miro if they want to integrate their research into the UxVs developed under the ALS and Cohort programs.

It is also recommended that Miro users read the paper "Software Engineering for Experimental Robotics"[2] to get a better idea of the design patterns used in DRDC Miro programs.

1.1 Before Getting Started

This document is a rough guide for users to begin programming within the Miro framework. Before starting, many different installations need to completed, including ACE, TAO and Miro. The ACE/TAO/Miro software can be very confusing and complicated at first glance, but to quote Douglas Adams: "DON'T PANIC." It actually isn't that tough.

In addition to this guide, it is also recommended that the user read the ULM Miro manual [3].

The first section of this document will take the reader through the rigorous process of installing ACE, TAO, Miro and all the other associated libraries required to get Miro to compile. The user must also understand that two branches of Miro are used at DRDC, the main branch from the open source community (labelled ulmMiro) and DRDC's branch (labelled drdcMiro).

1.2 Getting Started

This guide refers to Miro modules, but they may also be called services, processes or programs. All of these labels refer to a stand alone computer program within Miro which does a specific task such as a GPS or SICK laser driver, or a localization, mapping, or obstacle avoidance algorithm.

There are a series of files and classes associated with each Miro service. With experience, the user will begin to recognize key objects and functions. In addition, most of the Miro modules follow standard design patterns and the best way to understand them is by looking at code examples. It is expected that the user will refer back and forth between this guide and Miro code to gain a thorough understanding of the software.

A key first step in creating a new Miro module is defining what the module will implement. Almost every module will be based around sending or receiving data which is defined in a pre-established interface (IDL) file. Before creating a module one should follow these steps:

1. Answer the question: Which interfaces/services will I need to talk to (send or receive data from)?
2. Answer the questions: What do I want to do with this interface? Do I want to send or receive data? Do I want to be able to poll for data and call functions from other

services? Alternatively, do I want to have the other services notify me when data is ready?

3. Find a pre-existing example to base your code on, and adapt it to your needs. Find code that uses either the same interface (IDL), or handles data in a similar way (event driven vs. polled), or both. All Miro modules follow established design patterns, which simplifies reuse.

In order to facilitate the learning process, there are Miro examples in the directory called, drdcMiro/examples. If you wish to see something running, you can skip to Section 5 of this document to run the examples.

Before doing anything with Miro, you will need to have ACE and TAO installed, and the latest CVS version of the DRDC branch of Miro. In this manual, the install directory is referred to as *drdcMiro*.

Also note that this manual is available at drdcMiro/doc/als_miro_manual.

1.3 Background

This section will introduce Miro and its components.

1.3.1 What does Miro do?

Miro is an **component based framework** that achieves **network transparency**. These terms are defined below:

- **Component Based** Szyperski defines components as "binary units of independent production, acquisition, and deployment that interact to form a functioning system" [4]. Thus a component is an independent entity that is capable of executing without requiring the services of a complete system and is often considered to be a separate process that runs under its own workspace. Through the use of components a system can be decoupled into its constituent elements. This decoupling results in portable and modular software that exhibits *plug-and-play* characteristics.
- **Frameworks** are defined as a set of classes that embody an abstract design for solutions to a number of related problems. All code is implemented using standard design patterns allowing the reuse of preexisting code, through well defined interfaces (using the IDL language).
- **Network Transparency** is defined as the ability to send and receive data between computer processes which may or may not reside on the same machine, without having any knowledge of the underlying network structure.

Miro is a framework built upon the capabilities defined by the CORBA (Common Object Request Broker Architecture) standards. Miro uses the TAO (The ACE ORB) implementation of CORBA, and TAO is based upon ACE (Adaptive Communications Environment) middleware toolkit.

Miro embodies all of the toolsets, shown in Figure 1, into a set of object oriented design patterns. Knowledge of the design patterns will allow the user to take advantage of these toolsets without having to know specific implementation details.

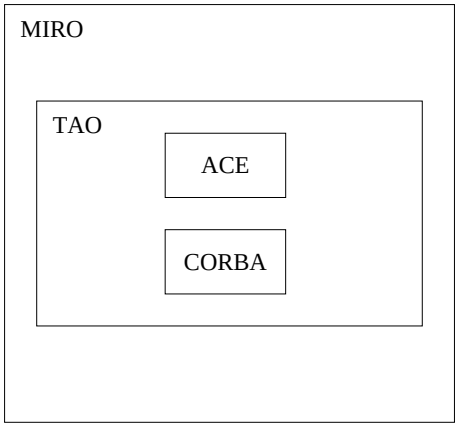

Figure 1: *The CORBA, ACE, TAO and MIRO relationship*

CORBA

- Provides distributed multi-threaded, multi-process programming.
- Automates many common network programming tasks.
- Provides services and infrastructure for passing objects over a network.
- Uses the Interface Definition Language (IDL) to define interfaces between programs.

ACE

- Library of Inter-process communication and system level calls.
- Allows platform independent programming.
- Provides the ACE Reactor Framework, which provides a flexible event handling mechanism (for use with hardware using TCP/IP and RS-232 protocols).

TAO

- A real-time CORBA implementation, based upon ACE.
- Designed for high performance and real-time communication.
- Provides a Naming Service for looking up and connecting to other modules.

1.3.2 What Else Do I Need To Know?

Miro is implemented using the C++ programming language and thus knowledge of object oriented programming is required. CORBA supports other language mapping such as C and Java; hence such languages could be used if required, but this route is not recommended as Miro uses only C++ at this time.

At the present time Miro runs only under Linux, thus a basic knowledge of using Linux/Unix is required. The Miro developer will require knowledge of the X Window system and the Linux/Unix command line. Since Linux is the operating system of choice, Miro also uses the standard Linux development tools such as: autoconf, the g++ compiler and the gdb/ddd debugger. Miro simplifies the use of these tools.

All Miro files are stored in the AISS CVS repository, which allows for the simultaneous development of software by multiple users. For those unfamiliar with CVS, there are several books available, including [5].

2 Install Instructions

It is assumed that the user is using a Redhat/Fedora based system with the system update program *yum* installed. If you are using another linux system you will have to install the following packages according to your particular distribution of linux.

2.1 Pre Installations

To install ACE/TAO the following lines are required in the .bashrc file in your $HOME directory. If you're using a different shell you may have to add it to a different file.

In your home directory edit the .bashrc file and add the following two lines:

```
source $HOME/.miroProfile
```

Then create the *.miroProfile* in your home directory as per the instructions in Section 2.1.1. Finally, source the .bashrc file by typing:

```
source ~/.bashrc
```

2.1.1 .miroProfile

The *.miroProfile* file should contain the following text:

```
# User specific variables for Miro
```

```
# Define the paths where ACE/TAO are located
export ACE_ROOT=/usr/local/ACE
export TAO_ROOT=$ACE_ROOT/TAO
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/ace:$ACE_ROOT/lib

# Miro Naming Service Variables, which includes multi-threading
export IP="YOUR_IP_ADDRESS"
export PORT="2809"
export NS="-m 0 -ORBEndpoint iiop://$IP:$PORT"
export NSC="-ORBInitRef NameService=iioploc://$IP:$PORT/NameService"
export MT="-ORBSvcCONF drdc.conf"

# Define the ATLAS variable
export ATLAS_ROOT=/usr/local/ATLAS
export ATLASARCH=Linux_P4SSE2_2

export NameServiceIOR=corbaloc:iiop:$IP:$PORT/NameService

#Define the path where MIRO is located
export MIRO_ROOT=PATH_TO_ULM_MIRO/ulmMiro
```

Please note that YOUR_IP_ADDRESS should be replaced with your computer's IP address and PATH_TO_ULM_MIRO should be replaced with the full path name to your ulmMiro installation. Additionally, if you have installed any of the packages in different places then you will have to change *.miroProfile* accordingly.

2.2 QT Installation

In order to compile ACE/TAO QT must be installed. Many distributions of linux come with QT. Below are QT installation instructions using the *yum* package manager:

```
yum install qt
yum install qt-devel
yum install kdelibs
yum install kdelibs-devel
```

For non *yum* users, the process can vary depending on the package downloaded. Please follow the instructions that come with the package you downloaded.

An additional OpenGL 3d package, **libQGLViewer** is used by some drdcMiro Qt based utilities and can be installed if desired. Though nonessential, these utilities permit viewing of vehicle frames and data sets in 3D:

1. Ensure Qt is properly installed.

2. Download **libQGLViewer** from http://artis.imag.fr/ Gilles.Debunne/QGLViewer/.

3. Follow the install instructions.

2.3 ACE/TAO Installation

In this document, it is assumed that the reader is using Fedora Core 3, GCC version 3.4 and ACE/TAO versions 5.4.8 and 1.4.8 respectively. The instructions follow below:

2.3.1 ACE Installation

1. Open a terminal and login as root.

2. Make the following two directories:

   ```
   mkdir /usr/local/ACEDIR
   mkdir /usr/local/ACE
   ```

   The first directory will hold the ACE/TAO installation tarball while the second directory will contain the ACE/TAO installation.

3. From your command line, export the three environment variables below:

   ```
   export MY_ACEDIR=/usr/local/ACEDIR
   export ACE_ROOT=/usr/local/ACE
   export LD_LIBRARY_PATH=$ACE_ROOT/ace:$ACE_ROOT/lib:$LD_LIBRARY_PATH
   ```

4. Download ACE/TAO 5.4.8 from http://deuce.doc.wustl.edu/Download.html in to "/usr/local/ACEDIR".

5. Download the install script from http://www.cs.wustl.edu/ schmidt/ACE-install.sh into "/usr/local/ACEDIR/".

6. Change directories into "/usr/local/ACEDIR/".

   ```
   cd /usr/local/ACEDIR
   ```

7. Change the permissions

   ```
   chmod 755 ACE-install.sh
   ```

8. Run the install script:

   ```
   ./ACE-install.sh
   ```

9. You'll then be asked several questions, for which the answers are provided after each question:

   (a) Question

   ```
   ACE source in
   ACE will be installed in /usr/local/ACE

   OK to continue? [Y/N] : \c
   ```

Answer
y

(b) Question
```
Save a copy of existing ACE installation? [Y/N] : \c
```
Answer
n

(c) Question
```
...
config-win32-visualage.h
config-WinCE.h

Type the filename for your compiler:  \c
```
Answer
config-linux.h

(d) Question
```
...
platform_win32_icc.GNU
platform_win32_interix.GNU

Type the filename for your compiler:  \c
```
Answer
platform_linux.GNU

(e) Question
```
Make ACE now with default setup? [Y/N] : \c
```
Answer
y

2.3.2 TAO Installation

1. Set and export a TAO_ROOT environment variable
   ```
   export TAO_ROOT=$ACE_ROOT/TAO
   ```
2. Make the code in the $TAO_ROOT directory
   ```
   cd /usr/local/ACE/TAO
   make
   ```

2.4 Pre-Miro Installation

2.4.1 Boost

The use of high-quality libraries such as Boost speeds initial development, results in fewer bugs, maximizes reuse, and cuts long-term maintenance costs. In addition, Boost libraries are becoming de facto standards which many programmers are already familiar with.

Install Boost using *yum*:

```
su -
yum install boost boost-devel
```

2.4.2 GSL

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License. The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

Unlike the licenses of proprietary numerical libraries the license of GSL does not restrict scientific cooperation. It allows you to share your programs freely with others.

GSL is a simple install if you have YUM:

```
yum install gsl
yum install gsl-devel
```

2.4.3 ATLAS

ATLAS is an optimized version of BLAS and LAPACK written in C that can perform very fast matrix calculations. As it probes the CPU to optimize its code, it makes different file folders to contain the name of the operating, processor type, and number of processors[2]. For example on the Raptor UGV computer ATLAS code compiles under Linux_P4SSE2_8 [3].

1. Download the latest ATLAS Package from
   https://sourceforge.net/project/showfiles.php?group_id=23725

2. Open a terminal, login as root, move the package to /usr/local/ and untar it, the example below assumes you downloaded version 3.6.0.

   ```
   su -
   cd /usr/local
   cp /<path>/atlas3.6.0.tar.gz /usr/local/
   tar -zxvf atlas3.6.0.tar.gz
   ```

3. Change directories to /usr/local/ATLAS, and build the code

   ```
   cd /usr/local/ATLAS
   make
   ```

4. At this point you will be asked a lot of questions. You should be able to just hit enter for all of them. **EXCEPT IF YOU ARE ASKED THE QUESTION BELOW**, you'll want to make sure to choose the appropriate processor type with respect to your machine for the following question:

[2] It can hyper thread matrix operations which will boost your code's performance.
[3] For 8 processor usage since the the Raptor computer has 4 hyperthreaded cores

```
Probing for architecture:
Enter your machine type:
1. Other/UNKNOWN
2.AMD Athlon
3. 32 bit AMD Hammer
4. 64 bit AMD Hammer
5. Pentium PRO
6. Pentium II
7. Pentium III
8. Pentium 4
Enter machine number [8]:
```

5. You will also be asked:

   ```
   Enter Architecture name (ARCH) [Linux_P4SSE2_2]:
   ```

   For which the answer is: Linux_P4SSE2_2 or whatever you put in your ".miroProfile" file.

6. After this is complete, you need to compile and install it

   ```
   make install arch=Linux_P4SSE2_2
   ```

2.4.4 LAPACK and LAPACK++

LAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations[4] are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision. LAPACK++ is a C++ implementation of LAPACK.

1. Login to a terminal as root and install the g77 Fortran compiler and LAPACK:

   ```
   su -
   yum install gcc-g77
   yum install lapack
   ```

2. Download the latest LAPACK++ package from
   http://sourceforge.net/project/showfiles.php?group_id=99696

3. Then as root install the package, the example below assumes you're using version 2.3.0:

   ```
   su
   tar -xvzf lapackpp-2.3.0.tar.gz
   cd lapackpp-2.3.0
   ./configure --with-atlas-libs=/usr/local/ATLAS/lib/Linux_P4SSE2_2/
   make
   make install
   ```

[4] LU, Cholesky, QR, SVD, Schur, generalized Schur

2.4.5 IEEE1394 Video Support

To install IEEE1394 ensure that the DAG *yum* repository is enabled and do the following:

```
su -
yum install libdc1394 libdc1394-devel libraw1394 libraw1394-devel
```

2.4.6 OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real time computer vision.

Example applications of the OpenCV library are Human-Computer Interaction (HCI); Object Identification, Segmentation and Recognition; Face Recognition; Gesture Recognition; Motion Tracking, Ego Motion, Motion Understanding; Structure From Motion (SFM); and Mobile Robotics.

Download ffmpeg version 0.4.9 and opencv version 0.9.7 from http://prdownloads.sourceforge.net/opencvlibrary/ and follow the instructions below:

```
yum install ffmpeg ffmpeg-devel
yum install opencv opencv-devel opencv-python
```

2.5 ULM Miro Installation

1. Make sure *.miroProfile* has been sourced as per Section 2.1.1.
2. Checkout the Miro code. In your projects directory type:

   ```
   cvs -d :ext:<username>@mao:/usr/local/cvsroot co -kk ulmMiro
   where <username> is your login name, e.g. sverret
   ```

3. Edit *.miroProfile* and change the following line to reflect where your ulmMiro code is stored:

   ```
   export MIRO_ROOT=/home/alsuser/Projects/ulmMiro
   ```

4. Re-source *.miroProfile*:

   ```
   source ~/.miroProfile
   ```

5. Use a terminal to browse to the directory that you downloaded Miro to

   ```
   cd ~/Projects/ulmMiro
   ```

6. Bootstrap the configuration process by typing:

   ```
   ./bootstrap
   ```

7. After Bootstrap runs you should see the word “autoconf” at the end of the script. If not, run autoconf

```
autoconf
```

8. Configure Miro using the following command.

   ```
   ./configure
   make
   ```

9. To compile optional Player [6] support use the –enable-Player configure option.

2.6 DRDC Miro Installation

1. Ensure that *.miroProfile* points the shell variable $MIRO_ROOT to the right location.

   Edit *.miroProfile* and change the following line to reflect where your ulmMiro code is stored:

   ```
   export MIRO_ROOT=/home/alsuser/Projects/ulmMiro
   ```

   and add a line identifying where you will install drdcMiro. This isn't necessary for the *configure* or *make* steps, but some drdcMiro executables look for this environment variable:

   ```
   export DRDC_ROOT=/home/myDirectory/Projects/drdcMiro
   ```

2. Re-source *.miroProfile*:

   ```
   source ~/.miroProfile
   ```

3. Now you are ready to install the drdcMiro code, which uses the ulmMiro code as a library.

4. Checkout the Miro code. In your projects directory type:

   ```
   cvs -d :ext:<username>@mao:/usr/local/cvsroot co -kk drdcMiro
   where <username> is your login name, e.g. sverret
   ```

5. Use a terminal to browse to the directory that you downloaded Miro to

   ```
   cd ~/Projects/drdcMiro
   ```

6. Bootstrap the configuration process by typing:

   ```
   ./bootstrap
   ```

7. After ./bootstrap runs you should see the word "autoconf" at the end of the script. If not, run autoconf:

   ```
   autoconf
   ```

8. Configure Miro using the following command:

   ```
   ./configure
   make
   ```

9. To compile optional Digiclops support use the –enable-Digiclops configure option.

3 Basic Concepts

3.1 IDLs (Interface Definition Language)

IDL is the computer language used to define the interface of a CORBA object. All communication within Miro has its structure defined in an IDL file. These files are located in the drdcMiro/idl directory. There are a number of different interfaces already defined and the number will continue to grow and change over time.

The IDL Language:

- Allows the developing of complex CORBA interfaces.
- Uses a C++ like syntax and is easy to use and understand.
- Is compiled into C++ using the TAO IDL compiler.
- Specifies data structures and functions which represent the communication between processes in the Miro system.
- Describes the interface only, not the implementation. The implementation is done by overloading the methods created in the IDL.

CORBA is not language specific, thus IDL is a language of its own, and is compiled by a special IDL compiler into a particular language (C++ in DRDC's case). IDL is a declarative language. In other words, IDL code contains no expressions or evaluations. In this way, IDL can best be thought of as a non language specific class header file. In fact, most C++ programmers will find the syntax of IDL familiar. In a typical class header file there is a declaration of a class and all of its data and method members. The syntactical expressions which perform the function of the class are generally reserved for the associated C++ file. Under Miro, the IDL file is analgous to the .h header file, and the actual module code is analgous to the .cpp file. The module code is generally referred to as an *Implementation* of the IDL.

More on creating and using IDLs can be found in Section 9 of this manual.

3.2 Naming Service

The Naming Service is the central point where all processes connect to find references to other processes or events in Miro. Every Miro process will need to connect to a Naming Service.

The Naming Service is run from the /usr/local/ACE/TAO/orbsvcs/Naming_Service directory by typing one of the following:

1. ./Naming_Service -m 1

   This will multicast the Naming Service allowing any Miro module on the same subnet to find it automatically when run. Note that if there are two network services (i.e eth0

and eth1) and you attempt to connect this way, it will take a long time. Similarly, if interface eth0 is active, but not plugged into the network, it will take services a long time to find the Naming Service.

2. ./Naming_Service -m 0 -ORBEndpoint iiop://127.0.0.1:2809

   This specifies a specific place for the Naming Service to be found, in this case on the loopback interface at port 2809. Miro processes attempting to use this Naming Service will need to be run with the following argument:

   ./exampleservice -ORBInitRef NameService=iioploc://127.0.0.1:2809/NameService

Chapter 3 of the Miro Manual describes the Naming Service in more detail.

3.3 Polled Mode Data Transfer

Polling is the most basic way of accessing data located within another Miro service. After a reference to the desired CORBA object in the Naming Service is found, modules make function calls and return values from them as if they were running in the same process.

The object being polled, must implement an interface of an IDL type. The polling service makes a function call as defined in the IDL and the data structure is returned to the calling process. Because the processes involved may not be running on the same machine, the IDL language doesn not permit pointers to be used as arguments or return values for any of these functions.

The implementation of the IDL interface is stored in the modules drdcMiro/src/ directory, or in the shared drdcMiro/src/miro directory. The former directory is common when the interface is specific to one service while the latter directory is used to store generic implementation for potential reuse by multiple modules. For example the implementation of the Raptor IDL which contains function calls specific to the Raptor UGV hardware is stored in drdcMiro/src/raptorCommand, while the Range3dSensor IDL implementation is stored in drdcMiro/src/miro, because many potential sensor services could conceivably produce events of this type.

This type of data interaction is useful for data transfers that occur infrequently, when data is to be polled from a service, or certain attributes are to be set in a service.

Chapter 4 of the Miro Manual further describes Polled Mode Data transfer.

3.4 Publish/Subscribe Event Driven Data Transfer

Miro also provides event driven data transfer using the CORBA Notification Service. In this context a service generates an event and publishes data to modules which have subscribed to the event. This supplier registers its event with the Event Channel in the Naming Service and the Event Channel is usually named *EventChannel*. The supplier gives its events a descriptive name within the event channel, such as *Imu*, *Gps*, or *Laser*. Then, each module

which wishes to subscribe to this data adds a subscription to the EventChannel. The consuming module can now receive these events using an EventHandler (callback function) which deals with these events when they occur.

The data structures for event data are defined in an IDL file, but are not actually part of the interface itself. This type of data transfer is good for asynchronous events, allowing a module to respond quickly to availiable data.

Chapter 4 of the Miro Manual covers this topic in more detail.

3.5 drdcMiro Directory Structure

There are a number of sub-directories within drdcMiro:

- Code is run from drdcMiro/bin.
- Miro module code is located in drdcMiro/src.
- IDLs are located in the drdcMiro/idl directory along with their compiled header files.
- Examples of code usage are found in the drdcMiro/examples directory.
- Generic IDL implementations are located in the drdcMiro/src/miro directory[5], while others are stored in the src directory of the service which implements them[6] as was discussed in Section 3.3.
- Utilities for displaying and debugging data are found in drdcMiro/utils.

4 Miro Utilities

4.1 LogPlayer/LogNotify

The Miro LogPlayer and LogNotify services provide a powerful method for testing and debugging Miro code. The LogNotify program records user specified events events and event data from an event channel to a log file.

The LogPlayer provides a GUI for replaying these logged data files. From the Miro module point of view there is no difference between events created using the LogPlayer and events from actual real-time services and hardware. This means that situations can be recreated/tested/debugged without the need to have the actual hardware present.

As an example, the file GPS_Run_Mar_21_05.log, located in the drdcMiro/utils/logPlayer directory, contains data and events logged from the GpsSokkia driver using the Raptor vehicle. Playing this file using the LogPlayer re-creates the *Gps* events, logged under the *EventChannel*, in time sequenced order. A consumer of *Gps* events can not distinguish

[5]Examples of IDL implementations are ImuImpl and GpsImpl
[6]The PlayerBasePlayerMotionImpl follows this format

between events produced by the LogPlayer and true hardware *Gps* events. The LogPlayer is limited by its inability to directly recreate data transactions where a Client polls for data.

These programs are found in the ulmMiro/utils/logNotify and ulmMiro/utils/logPlayer directories.

This topic is covered in more detail in the Chapter 11 of the Miro Manual.

4.2 nslist Naming Server Viewer

The nslist utility displays the events and objects registered in the Naming Service. It is located in the /usr/local/ACE/TAO/utils/nslist directory.

For a multicast naming service type: ./nslist

Otherwise, point nslist at the correct naming service: ./nslist -ORBInitRef NameService=iioploc://127.0.0.1:2809/NameService

4.3 Debug and Logging Services

Miro has built in facilities to assist in the debugging and logging of software. According to Chapter 10 of the Miro Manual:

> "The logging of debug, information and error messages can help a great deal in the recovery from failure situations. ... Unfortunately the sheer mass of console output from the different modules can quickly become complex. So *Miro* comes with a set of logging facilities that cover different levels of the system functionality."

The following line shows an example of a debugging statement:

```
MIRO_DBG_OSTR( MIRO, LL_DEBUG, "Map Size:" )
```

To enable this debug printout, command line options are appended to the involcation of the program as follows:

```
./ProgramName -MLL 7 -MLF MIRO
```

or

```
./ProgramName -MiroLogLevel 7 -MiroLogFilter MIRO
```

For example code that illustrates the capabilities of the Miro logging facilities look at the `ulmMiro/tests/log` directory which shows the usage of all the Miro debug levels and categories. See the Miro Manual for a complete description of the debugging and logging capabilities of Miro.

5 Miro Example Code

All Miro modules follow general design patterns where the full details regarding these design patterns can be found in [2]. For example, hardware, device driver modules use the Publish-Server and Reactor design pattern. Similarly, modules which subscribe to one type of event and create another type of event, follow the Publish-Subscribe design pattern. Four example services, as shown in Figure 2, have been created in the drdcMiro/examples directory which illustrate the use of design patterns in Miro services. All of the example files require the Example interface that is defined in drdcMiro/idl/Example.idl.

1. ExampleA is a server than can respond to polling requests.
2. ExampleB is a client that polls data from the ExampleA server.
3. ExampleC creates the *ExampleEventChannel* and publishes events of type *ExampleEvent* on this channel.
4. ExampleD subscribes to events of type *ExampleEvent* on the *ExampleEventChannel*.

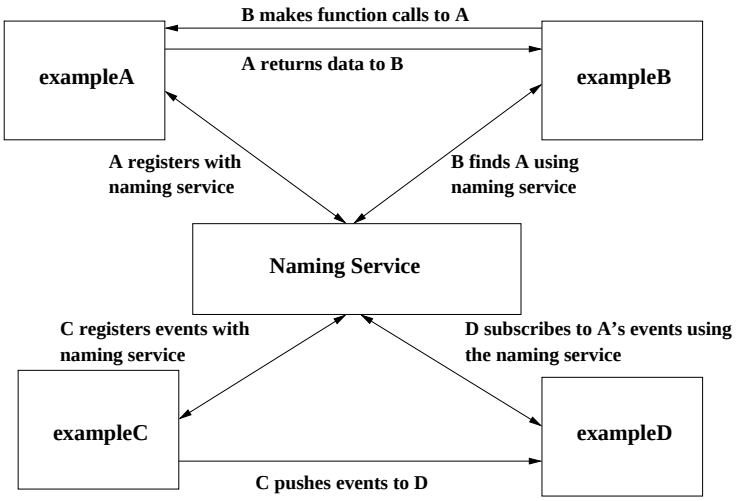

Figure 2: *Module interaction for example code*

The following sections describe these examples in more detail.

5.1 Client Design Pattern

The ExampleA and ExampleB modules illustrates the Client design pattern where a Miro client polls data from a Miro server. The ExampleA module, implemented under the Publish-Subscribe design pattern, acts as a server that implements the polled mode "Example" interface functions defined by the Example.idl. ExampleB, implemented under the Client design pattern, is a client that obtains an object reference to ExampleA and calling the polled mode method to retreive ExampleA data.

In order to use this example, do the following:

```
$cd /usr/local/ACE/TAO/orbsvcs/Naming_Service
$./Naming_Service -m 0 -ORBEndpoint iiop://127.0.0.1:2809
```

In a second terminal:

```
$cd drdcMiro/examples/exampleA
$export NSC="-ORBInitRef NameService=iioploc://127.0.0.1:2809/NameService"
$./exampleA $NSC -MLL 9 -MLF MIRO
```

In a third terminal:

```
$cd drdcMiro/examples/exampleB
$export NSC="-ORBInitRef NameService=iioploc://127.0.0.1:2809/NameService"
$./exampleB $NSC -MLL 9 -MLF MIRO
```

One can now see, from the output, that ExampleA has registered itself with the Naming Service. ExampleB has obtained a reference to ExampleA through the Naming Service, called the function *setData* to initialize internal variables, and finally called *getData* to continuously poll those values. As a client ExampleB will not be listed in the Naming Service. ExampleB increments the data each time it is polled; resulting in the output shown below:

```
The object 'ExampleA' has been found in the naming service
Variables a and b have been set to: 0 and 10
Retrieved variables a and b are: 1 and 11
Variables a and b have been set to: 2 and 12
Retrieved variables a and b are: 3 and 13
Variables a and b have been set to: 4 and 14
Retrieved variables a and b are: 5 and 15
Variables a and b have been set to: 6 and 16
```

Notice how the IDL is used: ExampleA includes the file ExampleS.h[7], and ExampleB includes the file ExampleC.h [8].

Note: MLL = Miro Log Level and MLF = Miro Log Filter.

[7]Example interface Supplier
[8]Example interface Consumer

5.2 Publish-Subscribe Design Pattern Example

ExampleC, implemented under the Publish-Subscribe design pattern, acts as a server creating an event channel in the Naming Service called *ExampleEventChannel* and publishes *ExampleEvent* data.

ExampleD, also implemented under the Publish-Subscribe design pattern, and uses an ACE Reactor timer event to create events at a user defined interval.

In order to run this example, do the following. It will run exampleC to produce an event every 1 second:

```
$cd /usr/local/ACE/TAO/orbsvcs/Naming_Service
$./Naming_Service -m 0 -ORBEndpoint iiop://127.0.0.1:2809
```

In a second terminal:

```
$cd drdcMiro/examples/exampleC
$./exampleC $NSC 1
```

At this point, exampleC show be indicating with its output that it is creating events. We now would like to subscribe to them. In a third terminal:

```
$cd drdcMiro/examples/exampleD
$export NSC="-ORBInitRef NameService=iioploc://127.0.0.1:2809/NameService"
$./exampleD $NSC
```

You will now see exampleD receiving data.

With either of the above two examples, you can use *nslist* to see the objects in the Naming Service by doing the following:

```
$ cd /usr/local/ACE/TAO/utils/nslist
$./nslist $NSC
```

6 Types of Files and Objects in Miro Code

Section 5 introduced simple Miro code examples, and now this code will be further dissected. All Miro modules will have their own directory in the drdcMiro/src/ directory. Within a modules directory, you will find a number of files, which create a number of classes. The relations between these Miro files and classes is shown in Figure 3.

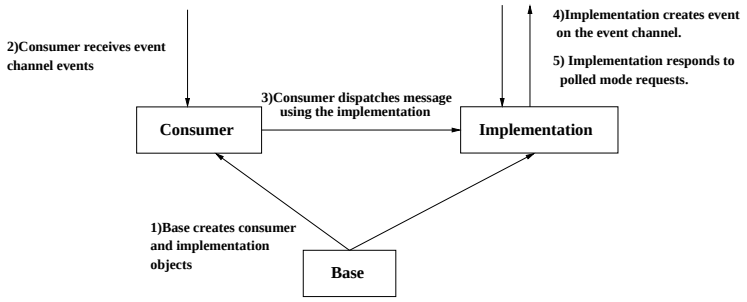

Figure 3: *A Typical Miro service*

6.1 Implementing the Publish-Subscribe Design Pattern

The basic files required to implement the Publish-Subscribe design pattern are the Base, the Consumer and the Implementation. Each of these files are discussed, in detail, in the following Sections.

Base

The Base is where the *main()* for the module is found. All the other files are included from here and the Base is responsible for creation of Miro Servers and Clients, Consumers and Event Handlers, ACE Reactors , along with all the other objects needed by the module. Finally, the Base runs in a loop where it handles events until it is terminated. An example of a Base file is drdcMiro/examples/exampleC/exampleCBase.cpp.

Consumer

The Consumer object, shown in Figure 4, processes all incoming messages. These may be messages coming from an Event Channel[9] or from an ACE Reactor[10]. It contains the heart of the overall module, and is where the the module responds to incoming events. An example of a Consumer file is drdcMiro/examples/exampleC/exampleCConsumer.cpp.

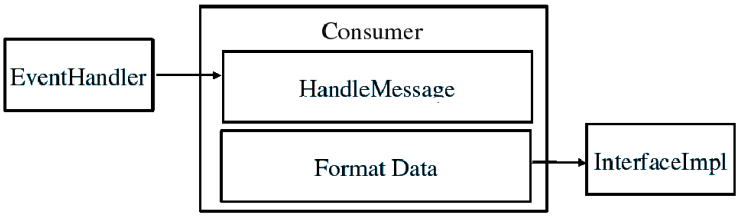

Figure 4: *A Consumer Object*

[9]If the Consumer is a StructuredPushConsumer object then event are delivered from other processes.
[10]If the consumer is a DevConsumer object then the events are delivered from a hardware device.

Implementation

Those modules which implement the CORBA interfaces to the IDL function definitions, will use a file of this type. It may be stored in the module's own directory, or found under the drdcMiro/src/miro directory. Usually an implementation will provide methods for both polling and asynchronously dispatching messages. The Implementation takes information from the Consumer, integrates it into the IDL interface, and publishes it on the event channel. An example of an Implementation file is drdcMiro/examples/exampleC/ExampleNotifyImpl.cpp.

6.2 Implementing the Publish Server and Reactor

Miro modules that implement device drivers, as shown in Figure 5 , require other files related to handling hardware devices. Details with respect to these device driver files are given in the following sections.

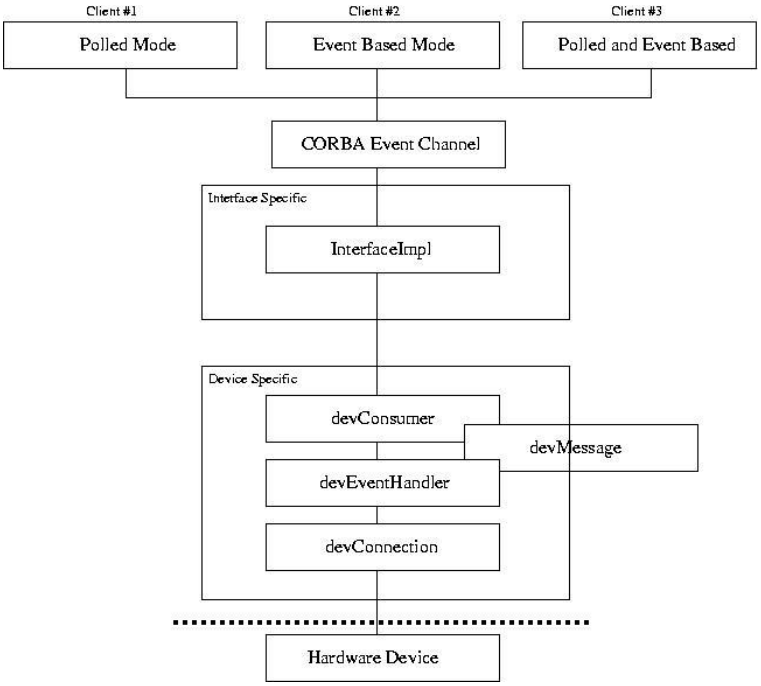

Figure 5: *Miro Device Driver*

Client Service

This is the lowest level interface to the hardware. It handles the setup and maintenance of the hardware communications[11] and is also responsible for sending the outgoing messages to the hardware. This class, shown in Figure 6, provides methods for sending/receiving communications over the physical medium. It uses an ACE Reactor Task to retrieve events on the specific file handle, and then sends these events to a callback routine within the Miro module. An example of a Client file is drdcMiro/src/gpsSokkia/GpsSokkiaClient.cpp.

[11] Examples of communications software/hardware are TCP/IP and ethernet, serial ports and RS-232

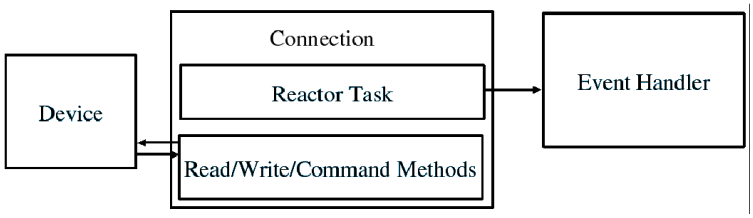

Figure 6: *Client Service*

Event Handler Service

The Event Handler, shown in Figure 7, responds to the incoming hardware events for a given file descriptor. Do not confuse these events with Miro EventChannel events, these are hardware generated events. The Event Handler creates a *Message* object, and processes the data stream into the Message format. The Event Handler remains responsive by performing very little internal processing; instead it passes the processing load to the Consumer of the data. An example of an Event Handler file is drdcMiro/src/gpsSokkia/GpsSokkiaEventHandler.cpp.

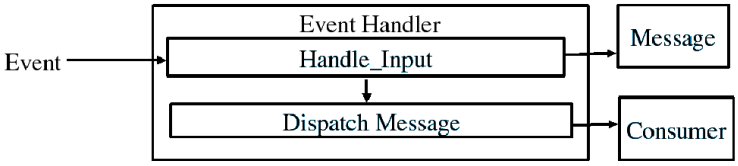

Figure 7: *Event Handler Service*

Message Service

This Message, shown in Figure shown in 8, stores incoming and outgoing hardware messages, until they can be handled by the Consumer. An example of a Message file is drdcMiro/src/gpsSokkia/GpsSokkiaMessage.cpp.

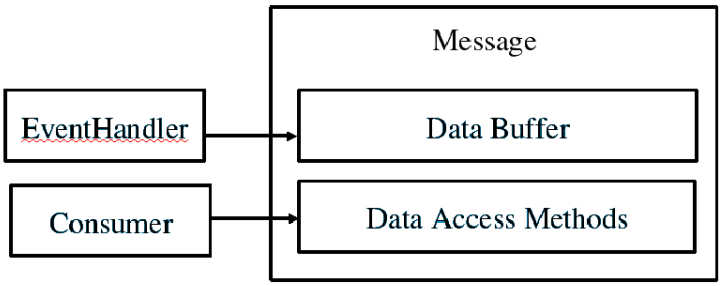

Figure 8: *Message Service*

.

6.3 Software Objects and Classes

6.3.1 Classes

Within the files mentioned above there are key classes and Miro concepts that the programmer needs to understand.

Miro::Server

The Server registers objects, events and event channels with the CORBA Naming Service[12]. After all the objects for the module have been declared, set up, and the references in the Naming Service found, the *server.run()* command is issued that puts the program in a loop; thus enabling the module to handle and dispatch events as necessary.

Miro::Client

The Client resolves the name of different objects registered with the CORBA Naming Service and if successful is able to poll data from a Server.

EventChannel_var

The EventChannel_var points to the event channel in the CORBA Naming Service that is used for publish/subcribe services. The *Server.resolvename()* function, if successfully resolved, will point EventChannel_var to an existing event channel. When required the *EventChannelFactory.create_channel()* function is called to create a new event channel. Usually only one event channel is required[13]; thus this function is only called once.

6.3.2 Objects

StructuredPushConsumer

This StructuredPushConsumer object responds to events from event channels, through the callback function *push_structured_event*. This function is called every time one of the subscribed events is received. It is passed a *StructuredEvent* that points to the event object containing data.

EventChannel_var notifyFactory

This object is used to create new event channels.

StructuredPushSupplier

The StructuredPushSupplier object pushes events and their data, at the CORBA level, onto the event channel. It is passed to the to the Implementation object which will use the StructuredPushSupplier to send the data.

[12]Remember, these are used by event driven services, as well as by services which implement an IDL interface.

[13]The Miro logging facilities can only synchronize data from a single event channel.

Implementation Objects

These objects package the outgoing data and implement functions declared in the IDL file. This includes remote methods, methods for polling data and in the case of event publishing, the *implementation.integrateData()* function that pushes data to the event channel.

AsynchDispatcher

AsynchDispatcher connects the Implementation object with the event channel, and publishes the events.

6.3.3 Objects For Hardware Interface Modules

ReactorTask

The ReactorTask is an ACE reactor which alerts the system to incoming hardware data by monitoring the hardware file descriptor. Upon the reception of data the ReactorTask passes the received data to the Event Handler.

DevConnection

DevConnection inherits from Miro::TtyConnection for serial communications or from Miro::SockConnection for TCP communications, and provides the setup for reading and writing to hardware.

DevEventHandler

DevEventHandler is based upond the ACE_Event_Handler and provides a callback function *handle_input* that reads the data from the hardware file descriptor. It then packages the data into the Message class to be processed by the Consumer. Finally, the *dispatchMessage()* function is called to alert the DevConsumer that there is a message ready to be processed.

DevMessage

DevMessage contains a buffer to hold data, and provides accessor methods to retrieve it.

DevConsumer

The DevConsumer, which is triggered when the DevEventHandler calls *dispatchMessage()*, processes the device specific data in DevMessage into an IDL compatible format using the *handleMessage()* function . This consumer is not to be confused with a StructuredPushConsumer, which processes Miro events from an event channel.

7 Discussion

With a basic understanding of the example code and the various software classes, components, and design patterns used to construct these examples, it is now possible to step through the examples in more detail. Also discussed in this section is an example that uses the Publish Server and Reactor design pattern to interface directly with a hardware device[14].

7.1 ExampleA - Implementation of a Server Module

Overview: ExampleA implements an IDL interface that allows other Miro modules to remotely invoke functions for polling; where polling is the sending and receiving of data between processes. The structure and form of the ExampleA implementation is shown in Figure 9

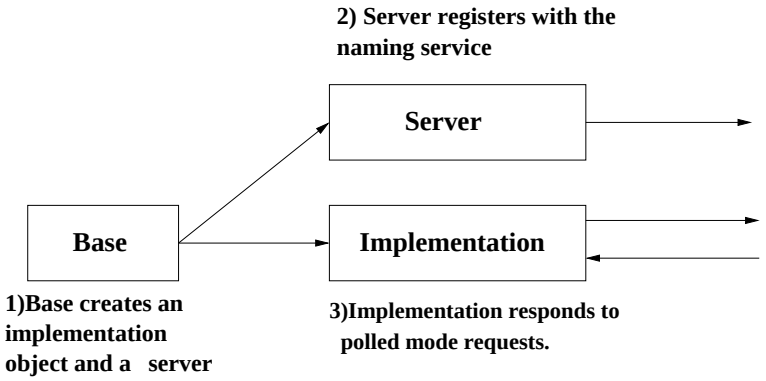

Figure 9: *ExampleA Structure*

Examples of this type:

- drdcMiro/examples/exampleA
- drdcMiro/src/3dmg
- drdcMiro/src/GpsSokkia

Typical Files:

- Base
- Implemplementation

Key Objects:

- Server

[14]To be specific this example is a device driver for a RS-232 serial connection with a GPS device.

- Implementation Object

Key Functions:

- *server.addToNameService()* - Registers the new object reference in the Naming Service

Description: In the Base file, the main() program creates a Miro::Server object. The server will be used to register objects with the Naming Service. It also creates an instance of the Implementation object, which responds to function calls from other Miro services.

The code for the Implementation object is found in the Implementation file. This Implementation object inherits from *POA_Miro::Example*, which means it will implement the functions for the interface *Example* that have been declared in Example.idl.

7.2 ExampleB - Polling Client Module

Overview: ExampleB polls a Miro Server modules using the defined IDL methods to retrieve/set data. The structure of this example is shown in Figure 10.

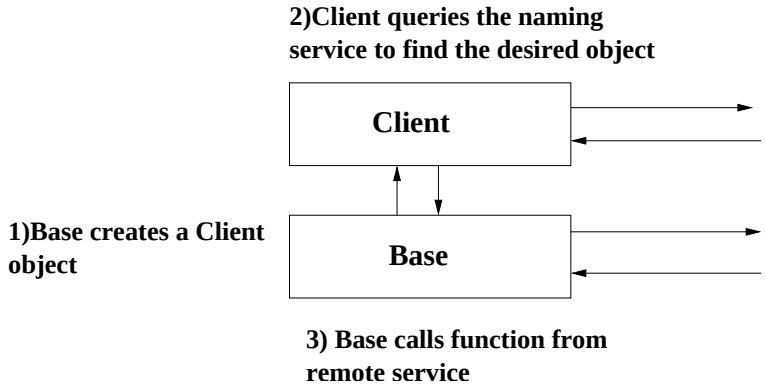

Figure 10: *ExampleB Structure*

Examples of this type:

- drdcMiro/examples/exampleB
- drdcMiro/examples/3dmg/3dmgPoll.cpp

Typical Files:

- Base

Key Objects:

- Client

Key Functions:

- *client.resolveName()* - Retrieves an object reference in the Naming Service.

Description: ExampleB creates a Miro::Client and uses the CORBA Naming Service to retrieve an object reference from the ExampleA server. This enables it to remotely execute ExampleA functions that, typically, allow the sending and receiving data structures via the structures defined in the IDL file.

7.3 ExampleC - Publisher of Events

Overview: ExampleC, derived from the Publish-Subscribe Server design pattern, asynchronous publishes events on the event channel. Modules subscribed to these events will automatically be notified every time an event is published. The structure of the Publish-Subscribe Server is shown in Figure 11. The data types, passed as events on the event channel, are define in the Example.idl file.

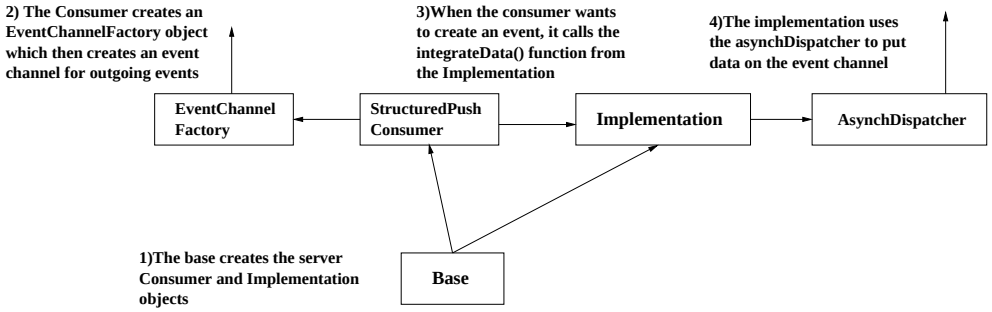

Figure 11: *ExampleC Structure*

Examples of this type:

- drdcMiro/examples/exampleC/
- drdcMiro/examples/notify/GenericNotify.cpp
- drdcMiro/src/3dmg/
- drdcMiro/src/GpsSokkia/
- drdcMiro/src/Laser/

Typical Files:

- Base
- Consumer
- Implementation

Key Objects:

- Server

- EventChannel_var
- StructuedPushSupplier
- EventChannelFactory_var
- Implementation Object
- AsynchDispatcher

Key Functions:

- *EventChannelFactory.create_channel()* - Creates new event channels in the Naming Service.
- *Implementation.integrateData()* - This function, located at the user application level, accepts and readies data, which is to be published as an event.
- *AsynchDispatcher.setData()* - This Miro level function causes the publication of the data to the event channel.

Description:

- The Base file creates a Miro server object that registers the channel and the events with the CORBA Naming Service. It also creates an ACE Reactor task which will function as a timer, so it will generate events at a specified frequency.

  The constructor for the Reactor Task (in the exampleCConsumer file) intializes the module such that outgoing events can be published to the specified event channel:

  - Instantiates the *notifyFactory* object, of type *EventChannelFactory_var*, which has the ability to create the event channel.
  - Creates an outgoing event channel object *ecout*, which is of type *EventChannel_var* using the *notifyFactory*.
  - Tells the StructuredPushSupplier which *EventChannel_var* points at the outgoing event channel.
  - Initializes the Implementation object with this StructuredPushSupplier.
- The Implementation object defines the *integrateData()* that publishes data to the event channel using the low-level Miro *AsynchDispatcher()* function.

7.4 ExampleD - Subscriber to Events

Overview: ExampleD, following the Publish-Subscribe Server design pattern, subscribes to events published by another Miro service. The structure of this module is shown if Figure 12.

Examples of this type:

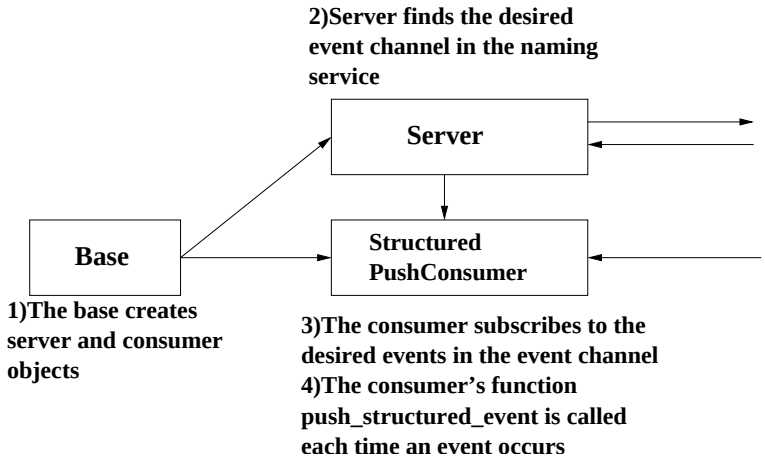

Figure 12: *ExampleD Structure*

- drdcMiro/examples/exampleD
- drdcMiro/examples/3dmg/3dmgNotify.cpp

Typical Files:

- Base
- Consumer

Key Objects:

- Server
- EventChannel_var
- PushStructuredConsumer

Key Functions:

- *PushStructuredConsumer.push_structured_event()* - This function is called every time a subscribed event is pushed to the event channel.
- *PushStructuredConsumer.setSubscriptions()* - This function facilitates the subscription to events registered with the CORBA Naming Service.

Description:

- The Base creates a server and then resolves the event channel, from which events will be received, into the *EventChannel_var*. It then creates a ExampleNotifyConsumer and passes it this *EventChannel_var* so that it can respond to events.

- Consumer - The Consumer inherits the *push_structured_event()* function which is called every time an event is published on the event channel. The constructor for the pushConsumer points the StructuredPushConsumer at the event channel defined by the *EventChannel_var*, and then uses the *setSubscriptions()* function to subscribe to named events.

7.5 Hardware Interfacing using the Publish Server and Reactor Design Pattern

Overview: This design pattern is commonly used to develop device drivers that interface with physical hardware. It uses the ACE Reactor to respond to events, generated via hardware interrupts, that signify the presence of new data. It then reads the external data, packages the data into an IDL defined structure and uses the Miro Server to publish the data to an event channel. The structure of device driver module is shown in Figure 13.

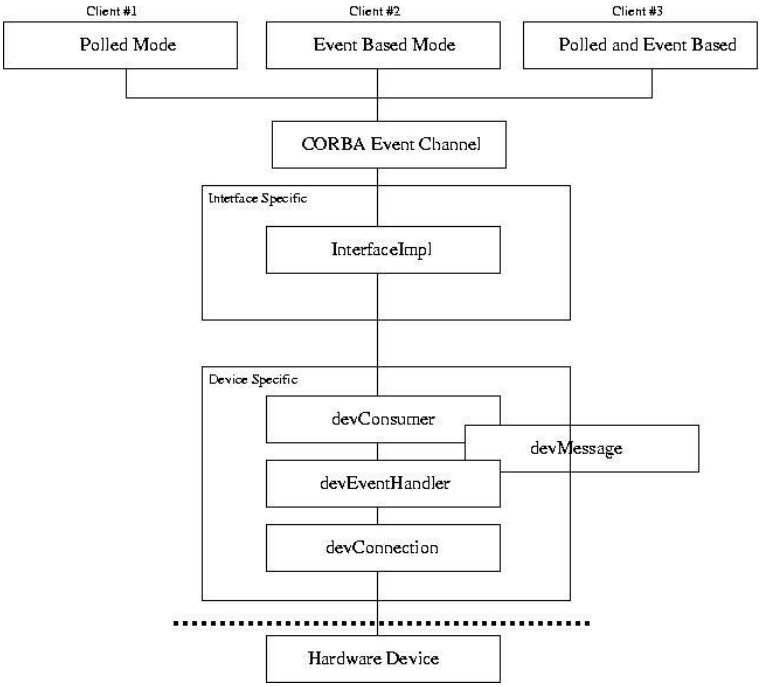

Figure 13: *Device Driver Structure*

Examples of this type:

- drdcMiro/src/3dmg
- drdcMiro/src/GpsSokkia
- drdcMiro/src/nodlaser
- drdcMiro/controlStation

Typical Files:

- Base
- Client(Connection)
- Event Handler
- Message
- Consumer

Key Objects:

- Base object which is a Miro Server
- ReactorTask
- Connection
- EventHandler
- Message
- Consumer[15]
- Implementation

Key Functions:

- *eventhandler.handle_input()* - This function is called by the ACE Reactor every time there is a hardware interrupt[16]
- *eventhandler.dispatchMessage()* - Alerts the Consumer that there is a new message available
- *consumer.handleMessage()* - This function is called every time that the Event Handler has created a new message
- *implementation.integrateData()* - This function sends data to the dispatcher class which publishes it on the event channel

Description[17]:

- The Base creates and initializes many more objects than in the previous examples. In fact, it also creates a base object to handle all of this, which none of the other examples do. The base object, which inherits from a Miro server, does the following in its constructor:

[15]This consumer is a DevConsumer, as opposed to the PushStructuredConsumer shown in Examples A through D.

[16]The interrupt could represent incoming data on the ethernet or RS-232 ports

[17]The drdcMiro/src/gpsSokkia is used in this example. This module interfaces uses a serial port to interface with a GPS device)

  - Creates objects for communicating with the hardware, such as the Connection, EventHandler, and Consumer objects.
  - Creates a Miro::ReactorTask object to respond to events on the serial port. This reactor object, and its Event Handler are then passed to the Connection object.
  - Calls a number of Connection object functions that initialize the GPS.
  - As in the event supplier examples above, it also sets up a StructuredPushSupplier, creates an outgoing event channel using EventChannelFactory and then adds the event type to the CORBA Naming Service.

- The Client is where the Connection object code lies. The connection inherits from type Miro::TtyConnection, which allows it to interface to the serial port. If the hardware in question used a TCP port, it would inherit from a Miro::SockConnection. The connection object has functions which can be called to send messages to the hardware, such as *setCommunication()* and *writeMessage()*. These in turn rely on the function call *ioBuffer.send_n()* to actually send data to the hardware. The equivalent call for a SockConnection is *peer_.send_n()*.

- Event Handler - The Event Handler's key function, *handle_input()*, is called by the Ace_Reactor every time an event occurs on the hardware file descriptor. The *handle_input()* function reads the message into the Message class and calls the *dispatchMessage()* function to alert the Consumer that a message is ready for processing.

- The Consumer *handleMessage()* function is called when there is a new Message created (i.e *eventhandler.dispatchMessage()* has been called). The Consumer then processes the message into the proper IDL structure and and calls the *integrateData()* function from the Implementation object to publish the event on the event channel.

- Message - For incoming data from hardware, the Message is filled with data by the Event Handler. Accessor methods are used by the Consumer to retrieve the Message data and process it.

7.6 Timed Event Publication

Sometimes a process, such as a localization filter, must create events at a specific rate, regardless of inbound event rates. Miro's basic Publication subpattern can achieve this result by inheriting from *both* from `StructuredPushConsumer` and ACE `ACE_Event_Handler` classes. While `StructuredPushConsumer` provides the event response logic, `ACE_Event_Handler` acts as an event handler for ACE timer events through the `handle_timeout(const ACE_Time_Value&, const void *)` method.

In ACE, timers are created and managed by an `ACE_Reactor` object. Thus the first step in making a timer is to either create an `ACE_Reactor` object or retrieve one. Fortunately, the Publish-Subscriber pattern has an `ACE_Reactor` object embedded in the Base [18] that can be retrieved through the `Miro::Server::reactor()` method. Using the

[18] The Base is derived from a Miro::Server and, therefore, contains an `ACE_Reactor` object.

`ACE_Reactor::schedule_timer()` call on the ACE reactor, one can create a timer and get a *timer ID* that uniquely identifies the timer to the reactor. Further, `schedule_timer()` accepts a `(void *)` argument that is passed to the event handler when the timer expires – providing the user with a simple mechanism of filtering multiple timers within the single event handling function,`handle_timeout()`

The `ACE_Reactor` object manages timers, issues events and automatically reschedules them when they expire. When the timer expires, the `ACE_Reactor` automatically calls `handle_timeout()`. This overloaded function can be structured to handle multiple timers simply by examining the `(void *)` (e.g. a pointer to an integer identifier) argument assigned during scheduling. The standard logic can then be used to publish events through the implementation. Note that measures (e.g. mutexes) should be taken to ensure overwriting or destruction of variables does not occur during the timer event handling period.

Readers are directed to the ACE documentation and examples such as `src/ platform/ PlatformConsumer.cpp` for more detail.

8 Adding a New Service to the Automake Structure

This section describes, in detail, the steps necessary to integrate a new service into Miro.

Automake is a set of tools that automatically generates the complex Makefiles associated with large source code projects. Although there are a few growing pains involved with using automake, once these are overcome the power of these tools becomes obvious. This section details how to a create a new Miro module using the *example* module as a guide. The *example* module will be fully integrated into the automake structure allowing new code to be compiled in the same manner as all other Miro code.

NOTE: While this tutorial is specific to Miro, much of the information learned can be transferred to other software packages which also use AutoMake

- **Create a new directory where the example service will be located**

  change directories to src directory of your Miro installation

  ```
  cd <path>/drdcMiro/src
  ```

- **Make a directory for the example driver**

  ```
  mkdir ex_service
  ```

- **Edit Makefile.am**

  Open drdcMiro/src/Makefile.am with your favorite editor and locate the section which looks as follows:

  ```
  raptordirs = \
          nodlaser \
  ```

```
        3dmg \
        gps \
        raptorCommand
```

This statement specifies the folders which contain services to be compiled for the Raptor Platform. You must add your directory name to this list.

NOTE: All directories except the final one in the list must be terminated with a "\" character to indicate that there is more information on the following line.

The new statement should look something like this:

```
raptordirs = \
        nodlaser \
        3dmg \
        gps \
        raptorCommand \
        ex_sevice
```

- **Write the source code for ex_driver**

  Change directories to ex_driver

  ```
  cd ex_service
  ```

  Create a file called main.cpp

  ```
  touch main.cpp
  ```

  Open main.cpp with your favorite editor and copy the following code into it:

  ```
  #include <stdio.h>

  int main ()
  {
          printf("HELLO WORLD\n");
  }
  ```

  NOTE: The focus of this tutorial is integrating with Automake, not writing a complete Miro service therefore the code in main.cpp will suffice:

- **Setup Makefile.am for ex_service**

  Create the file Makefile.am within the ex_service directory:

  ```
  touch Makefile.am
  ```

  Open Makefile.am with your favorite editor and copy the following code into it:

  ```
  ## -*- Makefile -*- ################################################
  ##
  ## This file is part of Miro (The Middleware For Robots)
  ##
  ## (c) 1999, 2000, 2001
  ```

```
## Department of Neural Information Processing, Ulm, Germany
##
## $Id$
## $Date$
##
##
##
#######################################################################

include $(top_srcdir)/config/Make-rules

bin_PROGRAMS = ExService

ExService_SOURCES = main.cpp

BUILT_SOURCES =
CLEANFILES = $(BUILT_SOURCES)

includefiles1 = $(ExService_SOURCES:.xml=.h)
includefiles2 = $(includefiles1:.cpp=.h)

ExService_LDADD = \
        $(top_builddir)/lib/libmiroSvc.a \
        -lmiroIDL \
        -lmiro

all-local:
        $(INSTALLPROGRAMS)

clean-local:
        $(CLEANPROGRAMS)
```

This file tells automake everything it needs to create the Makefile and built the program.

- bin_Programs = ExService
  This tells automake to create the executable ExService:
- ExService_SOURCES = main.cpp
  This tells Automake which sources to compile:
- ExService_LDADD
  This tells Automake which libraries are needed to compile.

In our case we only have one source file, main.cpp, which we need to compile. Whenever new source is added to a service it needs to be added to this file in order to compile it.

- **Edit configure.ac**

  Open drdcMiro/configure.ac with your favorite editor. Scroll down until you see the following:

```
AC_CONFIG_FILES([Makefile
                 idl/Makefile
                 src/Makefile
                 src/params/Makefile
                 src/makeParams/Makefile
                 src/abus/Makefile
                 src/b21Base/Makefile
                 src/base/Makefile
                 src/buttons/Makefile
                 src/can/Makefile
                 src/faulMotor/Makefile
                 src/faulTty/Makefile
                 src/image/Makefile
                 src/laser/Makefile
                 src/mcp/Makefile
                 src/miro/Makefile
                 src/msp/Makefile
                 src/panTilt/Makefile
                 src/pioneer/Makefile
                 src/pioneerBase/Makefile
                 src/playerClient/Makefile
                 src/playerBase/Makefile
                 src/nodlaser/Makefile
                 src/3dmg/Makefile
                 src/gps/Makefile
                 src/raptorCommand/Makefile
```

  Now add a reference to the ex_service Makefile:

```
src/ex_service/Makefile
```

- **Compile your code**

  You should now have everything in place and are ready to compile your new code. First we need to create a new configure file which will reflect the changes you have just made. Issue the following command:

```
./bootstrap
```

  Run the new configure script as follows:

```
./configure
```

  Finally compile your code:

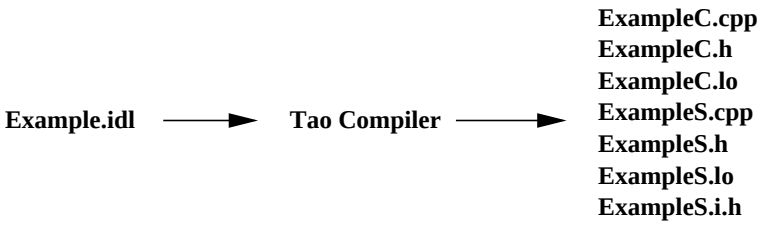

Figure 14: *The IDL Compile*

```
make
```

After compiling you should be able to cd to the ex_service directory and see the program ExService. Try running it, you should get the familiar output:

```
HELLO WORLD
```

9 OMG Interface Definition Language

With a solid understanding of Miro and the its software structure, it is now possible to write a new Miro service. A natural starting point for writing a Miro service is OMG's Interface Definition Language (IDL). IDL is a language used to define a CORBA object's interfaces. Given that CORBA is not language specific, IDL is a language of its own (similar to C++), which is then compiled by a special IDL compiler into a particular language binding.

Miro uses the TAO[19] IDL compiler to creates the C++ files required by programs that wish to communicate via the IDL interface. Figure 14 shows the files created by the IDL compiler for the C++ language binding.

The code listing, shown below, illustrates the similarities between IDL and C++.

```
//C++ code for a Person Class
struct person_traits
{
  float height;
  float waist;
  char  hair;
  int   age;
};

class Person
{
  private:
```

[19]Where TAO is The ACE Orb, which is an open source CORBA implementation.

```
    struct person_traits Traits;
  public:
    Person( float height, float waist, char hair, int age );
    float    getHeight();
    float    getWaist();
    char     getHair();
    int      getAge();
    person_traits getTraits();

};

//IDL code for a Person Class
struct PersonTraitsIDL
{
  float height;
  float waist;
  char  hair;
  long  age;
};

interface Person {
  void      setTraits(in float height,in float waist,in char hair,in long age);
  float     getHeight();
  float     getWaist();
  char      getHair();
  long      getAge();
  PersonTraitsIDL getTraits();
};
```

As can be seen, an IDL file bears a strong resemblance to a C++ header. Both define a structure which holds a person's traits that the class will store. Both define a number of accessor methods, which can be used to set/get the traits of the object. The main differences between the two files are that the IDL defines interfaces not classes. In addition, parameters are declared with in/out/inout direction indicators defining the parameter as an input, output, input/output.

The advantage of IDL is that one is able to write one interface to a CORBA object and compile it into numerous binding including: Java, C, C++, COBOL, etc. It is language independent, allowing interoperability between different languages. Another advantage is that IDL effectively promotes the object oriented paradigms to be used by languages which don't support Object Oriented principles such as C. Finally, IDL provides mechanisms that allow programs to share data across a network with relative ease.

IDL interfaces are used to create all of the interfaces between different software modules in Miro. An IDL compiler is then used to map the IDL to a specific language. In our case

we use the TAO IDL compiler that compiles to C++ code. All of the IDLs are defined in a single directory under Miro which can be found in drdcMiro/idl. The following examples show idl developed for the ex_service module presented in Section 8.

- **Change directories to drdcMiro/idl**

  ```
  cd drdcMiro/idl
  ```

- **Create a file for for the new IDL**

  ```
  touch myExample.idl
  ```

- **Open myExample.idl for editing**

  Copy the following code into myExample.idl

  ```
  #ifndef myExample_idl
  #define myExample_idl

  module Miro
  {
          struct myExampleIDL
          {
                  char Buffer[30];
                  long cmdLength;
          };
          interface myExample {
                  myExampleIDL getString();
          };
  };

  #endif
  ```

  This IDL declares an interface *myExample* with one member function *getString()*, which returns a structure called *myExampleIDL*. This structure contains a character buffer that will hold a string and a long that defines the length of the string.

- **Open drdcMiro/idl/Makefile.am for editing**

  Scroll down to the portion of the file which defines the variable globalsources and add myExample.idl:

  ```
  globalsources = \
          Raptor.idl \
          Time.idl \
          Exception.idl \
          WorldPoint.idl \
          WorldVector.idl \
          Position.idl \
          Velocity.idl \
  ```

```
MotionStatus.idl \
Odometry.idl \
Motion.idl \
ClosedMotion.idl \
SynchroMotion.idl \
DifferentialMotion.idl \
Stall.idl \
myExample.idl
```

Now type make and the new idl should compile. The following files will be generated by the TAO IDL compiler:

```
myExampleC.cpp  myExampleC.h    myExampleC.i.h  myExampleC.lo
myExample.idl
myExampleS.cpp  myExampleS.h    myExampleS.i.h  myExampleS.lo
```

These files represents the glue that allows your client process to obtain remote object references from other processes. The actual implementation of this example will be discussed in the next section.

9.1 IDL CORBA Sequences

IDLs can use structures called sequences to pass information between processes. These are variable length arrays, similar to C++ STL vectors, used to pass data between modules[20]. They are indexed much like arrays, except that they have special helper functions. For example, you can use sequence.length() to set or retrieve the length of the sequence.

9.1.1 Declaration of Sequences in an IDL

Within the IDL file "drdcMiro/idl/Example.idl" you will see such a structure:

```
struct ExampleStructIDL // Declaration of a simple structure
{
        double a;
           double b;
        };

// This defines a type of sequence, or variable length array,
// of the first structure
typedef sequence<ExampleStructIDL> ExampleListIDL;

// A higher level structure which contains an instance of the sequence
// and another variable
struct ExampleGroupIDL
```

[20]Note: Pointers and dynamic C++ STL constructs cannot be used with an IDL.

```
{
int c;
ExamleListIDL PointList;
};
```

9.1.2 Using CORBA Sequences within Modules

The module, which produces the data to fill in the structure, first sets the sequence length and then populates the data structure:

```
// Declares a structure of the type in the IDL
ExampleGroupIDL examplegroup;
examplegroup.c = 11;

// Set the length of the sequence in the structure
examplegroup.PointList.length(2);

// Fill in the values contained in the sequence
examplegroup.PointList[0].a = 22;
examplegroup.PointList[0].b = 33;
examplegroup.PointList[1].a = 44;
examplegroup.PointList[1].b = 55;
```

At this point, the Miro service can publish this data to the event channel. In order to retrieve the data, the receiving program might use something like the following:

```
// Declares a structure of the type in the IDL
ExampleGroupIDL receivedgroup;

// This retrieves the length of the sequence
int counter = receivedgroup.PointList.length();

// For example,
// the sequence data has some simple operations performed on it
for (int i =0; i < counter)
{
++receivedgroup.PointList[i].a;
double test=receivedgroup.PointList[i].a*receivedgroup.PointList[i].b;
}
```

10 Parameter Files

Miro uses the `makeParam` utility to build default parameter sets for a module from an XML file. An XML parser is also generated from the XML file which allows parameters to be loaded into the module at runtime from a seperate XML file. This allows the developer to make changes to the software configuration without having to recompile the software.

The essential steps to building a set of parameters and a parser are:

1. Construct a *compile-time* Parameter.xml file which declares any parameters used by the module and assign default values to them.
2. "Compile" this file using `makeParams` to generate `Parameters.h` and `Parameters.cpp` files. This can be setup in `Makefile.am` so that it occurs when your application is built.
3. Build your application.
4. Construct a run-time parameter file called <hostname>.xml. This file holds the necessary parameters for your process.

The following example illustrates the above steps:

To represent geometry on a vehicle, it is necessary to describe the position of many devices, some of which will be repositioned between trials. An XML configuration file is used to set up the location of these devices using a 3x1 position vector. This could be accomplished using three individual parameters, one *vector*, a *set* composed of three entries, or a *string* holding three numbers. Here we use a vector.

10.1 Compile-Time Parameter Files

Compile-Time parameter files describe the parameters that your process would like to exploit within Run-Time files. Basically, the compile-time file and `makeParams` creates code that will automatically parse XML and place the results into C++ structures.

The structure of the compile-time parameter file is typical XML. Consider the example Parameter.xml file:

```
1 <!DOCTYPE xml SYSTEM "PolicyEditor_behaviour.dtd">
2 < config>
3     <config_global name="namespace" value="Test" />
4     <config_global name="include" value="miro/SvcParameters.h" />
5     <config_global name="include" value="vector" />
6     <config_group name="test">
7       <config_item name="My" parent="Miro::Config" instance="true" >
8          <config_parameter name="TestVector" type="std::vector&lt;int&gt;" />
```

```
9           <config_parameter name="TestInteger" type="int" default="1"/>
10        </config_item>
11     </config_group>
12 </config>
```

Parameters are grouped through opening and closing symbols. The config file opens with a document type header, typically something like line 1 above. This is followed by the configuration data block, always within the `<config></config>` pair. Note that a 'slash' denotes the closure of an XML block.

The configuration data is used to configure the structure of C++ source files, so the next block is a series of statements for `namespace` declarations and `#include` files. To work with Miro's parameter system, one must include `SvcParameters.h`[21]. If either Sets or Vectors are being used, their headers must also be included. Since `makeParams` can be instructed to read *and* output filenames of the users choice, there is no need to place all parameter definitions into a single file. Modular Parameter definitions can be generated and included much like C++ headers (e.g. `SvcParameters.h` is machine generated).

The next block describes the data to be used at run-time and is contained within a `<config_group></config_group>` pair. The config group name on line 6 becomes the 'Section' name in the run-time XML file.

Line 7 describes the `config_item` name, parent and instance.

name is the name of the Parameter class you will make and will be appended by "Parameters". In this example, the compiled XML file will produce Parameters.h that contains a class called `MyParameters`, that will ultimately hold the parameters read at run-time. Dropping the name simply produces "Parameters" as the class name. This is handy, because it means you have some control over Parameter Class naming and can generate multiple different parameter parsers within the same XML compile-time file.

parent identifies the parent class of the parameter class you are defining. To work in the Miro system, it must *ultimately* inherit from `Miro::Config`. Of course you must include the necessary headers, too, hence the inclusion of `SvcParameters.h`.

instance is a boolean value (i.e. "True" or "False") that determines whether Miro should permit multiple instances of this parameter set ("False") or simply permit multiple use of a single instance("True"). In general, setting this to "True" is the safest strategy.

Line 8 shows the use of the "default" field in the creation of the TestInteger parameter. Any time this value does NOT appear in a run-time file, the value of `config.testInteger` will be 1. Note that the `config_item` identifier of Line 7 is closed at the end of the parameter definitions in typical XML fashion in Line 10.

[21] SvcParameters contains a series of "service" parameter types and ultimately includes `Miro::Config`

Finally, the parameter definitions follow `config_parameter` with name, type and others, (see the Miro manual):

name is the name of the Parameter. If `config_item name` is the name of the class described by this file, `config_parameter name` is the name of the data member within the class.

type identifies the type of parameter, usually simple types (e.g. double or int) see the Miro manual for the full list. However it is possible to declare more elaborate structures as is done in this example.

The parameter item, group and block sections are then closed as mentioned earlier.

10.1.1 Helpers

There are limits to this compile-time system, however. Creating default values for some data structures pose problems for `makeParams`. In such cases, `makeParams` parses a `<constructor></constructor>` pair that contain C++-like source code assigning default values. A good example of this approach can be found in the compile-time Parameter file `maps/terrain/Parameters.xml` and the helper `src/miro/MapDescriptionHelper.cpp`. Helpers are a stop-gap solution and can be difficult to build without an existing example.

For example, suppose you wish to assign a vector of integers with default values. Using traditional makeParam syntax this is impossible, however using a Helper file and a `<constructor></constructor>` pair, this becomes possible.

10.2 Parsing Run-Time Parameter Files

Once the compile time XML file has been generated and the module compiled, the default parameters can be overridden by passing a *run time* XML file to the module. This is done using the following code:

```
1 Test::MyParameters * params = Test::MyParameters::instance();
2
3 try
4 {
5   Miro::ConfigDocument * config = new
        Miro::ConfigDocument(argc, argv);
6   config->setSection("test");
7   config->getParameters("Test", *params);
8   delete config;
9 }
10 catch (const Miro::CException& e)
11 {
```

```
12   cerr << "Miro exception: " << e << endl;
13   rc = 1;
14 }
```

Line 1 declares an instance of `MyParameters` using the `instance()` operator, available to us since we declared `instance= "True"` in the XML file. In the try block, The parameter file is read into the `config` variable and pointed at the portion of interest ("test"). The parameters are then loaded into the variable `params` in line 7. Finally, the config variable is deleted once the parameters have successfully been read.

Note: the default filename which the module looks for is <hostname>.xml. Other files can be used using the "-MiroConfigFile" command line option (See the Miro Manual).

From this point on the variable `params` contains the parameters we want.

Note: All parameter names must begin with a captial letter in the runtime XML file while all parameter names in Parameters.h will begin with a lowercase letter regardless of how they were defined in the XML file. In practice it is best to always capitalize the first letter in both XML files.

10.3 Run-Time Parameter Files

After building the code and compiling, it is time to construct a run-time XML file such as the following:

```
1<config>
2  <section name="test">
3    <parameter name="Test">
4      <parameter name="TestVector">
5        <parameter valueA="2" />
6        <parameter valueB="4" />
7        <parameter valueC="2" />
8        <parameter valueD="5" />
9      </parameter>
10    </parameter>
11  </section>
12</config>
```

The structure is very similar to the compile-time version, but carefully note the grouping of parameters with matching open/close statements. Furthermore, the end bracket of line 4 has no slash, meaning *the parameter expression is not terminated*. For both Vectors and Sets, the `<parameter name...>` entry is followed by as many `<parameter value=" "/>` expressions as necessary. Note that, without exception, all parameters values are enclosed in quotes in the XML file.

10.4 Nested Data Structures

Datatypes such as `int`, `double`, `std::string`, `vector`, and `set` are useful for most situations, but sometimes it is necessary to structure parameters further. Nested structures can be built through careful use of compile-time XML files to accomodate these situations. Consider the example below (the headers have been removed for space):

```
<config_item name="Matrix"  parent="Miro::Config" final="false"  >
<config_parameter name="Row1" type="std::string" />
<config_parameter name="Row2" type="std::string" />
<config_parameter name="Row3" type="std::string" />
</config_item>
<config_item name="BodyFrame"  parent="Miro::Config" final="false" >
<config_parameter name="FrameName" type="std::string" />
<config_parameter name="HTransform" type="MatrixParameters" />
</config_item>
<config_item name="Body" final="false" parent="Miro::Config"  >
<config_parameter name="BodyName" type="std::string" />
<config_parameter name="HTransform" type="MatrixParameters" />
<config_parameter name="Frames" type="std::vector&lt;BodyFrameParameters&gt;" />
</config_item>
```

In this example three structures have been created, a `Matrix`, a `BodyFrame`, and a `Body`. Note that the `BodyFrame` contains a `Matrix` while the `Body` contains both a `Matrix` and a vector of type `BodyFrame`.

Some important points to note:

- all structures inherit from `Miro::Config`.
- a type with a `config_item` of *name* becomes *name*Parameters when nested in other structures.
- nesting can be of arbitrary depth.

The greatest difficulty with nested parameters lies in the run-time file construction, which can easily become confusing as illustrated in the following example which creates a Body name Raptor with two BodyFrames:

```
<parameter> <!-- Body -->
<parameter name="BodyName" value="Raptor"/> <!-- BodyName -->
<parameter name="HTransform" > <!-- HTransform -->
<parameter name="Row1" value="1.00 0.00 0.00 0.00" />
<parameter name="Row2" value="0.00 1.00 0.00 0.00" />
<parameter name="Row3" value="0.00 0.00 1.00 5" />
```

```
       </parameter> <!-- end of HTransform -->
<parameter name="Frames" > <!-- FrameList -->
       <parameter> <!-- BodyFrame -->
<parameter name="FrameName" value="CG"/> <!-- Frame -->
<parameter name="HTransform" > <!-- HTransform -->
<parameter name="Row1" value="1.00 0.00 0.00 0.00" />
<parameter name="Row2" value="0.00 1.00 0.00 0.00" />
<parameter name="Row3" value="0.00 0.00 1.00 0.00" />
       </parameter> <!-- end of HTransform -->
       </parameter> <!-- end of BodyFrame  -->
       <parameter> <!-- BodyFrame -->
<parameter name="FrameName" value="CabRailCentre"/> <!-- Frame -->
<parameter name="HTransform" > <!-- HTransform -->
<parameter name="Row1" value="1.00 0.00 0.00 0.00" />
<parameter name="Row2" value="0.00 1.00 0.00 0.00" />
<parameter name="Row3" value="0.00 0.00 1.00 5.00" />
       </parameter> <!-- end of HTransform -->
       </parameter> <!-- end of BodyFrame  -->
          </parameter> <!-- end of FrameList  -->
</parameter> <!-- end of Body  -->
```

Comments, using the angle bracket notation `<!-- comment here -->` can be used to clarify the XML structure. Great care must be taken to ensure list types are properly opened (i.e. "no slash on the parameter") and closed. Note that the `Matrix` type uses `std::string` instead of vectors of doubles. A major problem with the Miro Parameter system is the dependence on quoted parameter values. This forces each vector or set element to be a single parameter entry and makes XML parameter files very large and difficult to read. In this case, the solution was to collapse rowvectors into a single string, parsed later in the C++ source.

11 Using Qt and Miro

Qt is a cross-platform graphical user interface library sold commercially by Trolltech, but available for noncommercial use for free. This manual will not cover how to construct Qt interfaces, but will address some of the "mating" issues common to Qt and Miro. Some of these issues are addressed within the original Miro manual.

Just as Miro has a limited number of design patterns, Qt and Miro generally use two design patterns. Miro provides two basic services through its IDL interfaces, Polling and Publishing. To present Polled data graphically in Qt simply requires placing the correct Miro polling code within the appropriate event loop (e.g. when a button is pressed). Miro's native Qt examples fully capture this method and the reader is directed to study both the Miro Manual, the source code and `make` files for guidance and insight.

A more difficult problem involves presenting *events* to the Qt interface. Like any interface engine, Qt runs an internal *event loop* that services graphical interactions such as a buttondown event, slider event, or other widget event. Miro's `Consumer` runs a similar event loop to service inbound subscribed events on the eventchannel. To permit these two loops to function requires two steps:

1. create a Qt Process that, in turn, creates and `detach`es a Miro consumer into a separate thread.

2. create a Qt Event that invokes a Qt `customEvent()` method to update the interface.

The reader is directed to the `qtEventImu` application in the `utils/qtEventImu` directory. This simple application subscribes to Imu events and updates an interface as each event arrives. Some special notes:

1. Possibly due to unexpected deletion of resources in the Miro thread, CORBA types do not cross the Miro/Qt boundary without corruption. In any case, the solution is to copy the event contents into a substitute type and passing this allocated type to Qt.

2. The 'connection' between the Qt and Miro threads is made through a call to the `QApplication` object's `postEvent()` method. This method, in turn, calls the overloaded `customEvent()` method that contains the actual UI update code.

3. If events come in too quickly for Qt to process (a remote possibility), then it may be necessary to pull the `postEvent()` method out of the `push_structured_event()` handler and use a timer to house this call instead. See the "Timed Event Publication" section for more detail.

11.1 QtDesigner with Miro

Not surprisingly, hand coding Qt can be tedious, making Trolltech's QtDesigner UI builder software very tempting. This section describes how to use Designer within the Miro framework and assumes the reader is familiar with the Designer package and output file structure.

Qt relies on `qmake` to build makefiles and ultimately compile QtDesigner generated code. Unfortunately, there is no easy way of forcing the `automake` tools and `qmake` work together seemlessly. However, QtDesigner's file structure is simple enough that it can be forced to live within Miro's `automake` environment.

To keep things neat and tidy, put all graphical elements in GUI subdirectory of your application.

1. In your source directory, create a GUI subdirectory.

2. Using QtDesigner, create a project in the GUI directory. Build your GUI as you see fit, modifying all the *.ui.h components as necessary.

3. Add all the necessary includes to the .ui.h files if necessary, but don't bother with libraries.

4. Save the project into the GUI directory which will then contain a *.pro, *.ui and, possibly, *.ui.h files.

5. At this point the interface can be compiled into the Miro structure two ways, by hand or through abuse of *qmake*.
   (a) To manually create the *.h and *.cpp files from *.ui:
      i. type the commands:
      ```
      uic <myForm>.ui -o <myform>.h
      uic <myForm>.ui -i <myform>.h -o <myform>.cpp
      ```
      ...where <myform> is the name of your form. These will create the *.h and *.cpp files from the *.ui file
   (b) ...OR you can abuse *qmake* to achieve the same thing, albeit contained in a hidden directory:
      i. qmake the project to construct the 'Makefile'
      ii. make the project to generate the necessary .cpp files. Make will try, but fail, to compile the code. That's OK, because all you really wanted were the .h and .cpp files in the (hidden) .ui directory anyway. To see the hidden directories Qt Designer has made, type:
      `ls -a`

6. A satisfactory UI can be incorporated into Miro using the standard Makefile QT template (put *.h under sources and *.cpp under tomocsources) being sure to use the correct relative path locations.

Remember to include all the contents of the hidden .ui subdirectory in any CVS version of the GUI directory.

11.2 Contributing QtDesigner Widgets to the Widgets Library

If you want to contribute widgets to the "widgets library" then:

1. Open the *.cpp files (in the .ui directory if you used the qmake method) and hunt for all references to *.ui.h files.

2. Cut/Paste the contents of these *.ui.h files into the .cpp files. [22]

3. Copy <myform>.cpp files and <myform>.h headers to utils/widgets, but NOT <myform>.moc.cpp or moc_<myform>.cpp files. These will be autogenerated from the <myform>.h files during the widget make.

4. Add the filenames to the Makefile.am (put *.h under sources and *.cpp under tomocsources) and type:
   ```
   make
   ```

[22] Technically these first two aren't necessary, but then you'd have to copy the *.ui.h files into the widgets directory, too.

References

[1] Broten, G., Monckton, S., Giesbrecht, J., Verret, S., Collier, J., and Digney, B. (2004), Towards Distributed Intelligence - A high level definition, (DRDC Suffield TR 2004-287) Defence R&D Canada – Suffield.

[2] Broten, G., Monckton, S., Giesbrecht, J., and Collier, J. (2006), Software Engineering for Experimental Robotics, Number DRDC Suffield SL 2005-227, Ch. UxV Software Systems, An Applied Research Perspective, Springer Tracts in Advanced Robotics.

[3] Department of Computer Science, University of Ulm (2005), Miro Manual, 0.9.4 ed, University of Ulm, Ulm, Germany.

[4] Szyperski, C. (1998), Component Software: Beyond Object-Oriented Programming, Addison-Wesley, Reading, MA.

[5] Vesperman, Jennifer (2003), Essential CVS, First edition ed, O'Reilly & Associates, Inc., Sebastopol, CA.

[6] Gerkey, Brian, Vaughan, Richard T., and Howard, Andrew (2003), The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems, *Proceedings of the 11th International Conference on Advanced Robotics, University of Coimbra, Portugal*, pp. 317–323.

Annex A: CVS Protocol

Both DRDC Miro and ULM Miro are labeled as separate modules in the Defence R&D Canada – Suffield Concurrent Versioning Software (CVS) system. They are labeled drdcMiro and ulmMiro respectively. Instructions on checking out sandboxes of these repositories is listed in Sections 2.5 and 2.6. However, in order for drdcMiro to compile correctly the proper version of ulmMiro must be updated and compiled. Below are the instructions for a user that has access to the Miro code directly from ULM to update their code, import it into DRDC's version of the ULM code along with tagging, labeling, merging the appropriate repositories so that drdcMiro will still compile with ulmMiro.

The ULM code has already been imported into the Defence R&D Canada – Suffield CVS repository, therefore, the following three steps are unnecessary. However, if you were importing code for the first time you would do the first two steps and every time you wanted to check out a fresh sandbox you would do the third step.

1. Import the initial release from the vendor into our CVS repository. In the example below we're importing from the vendor vendorName the release named vendorRelease_v1_0 into the project users/sverret/testProject. You must be in the directory you want to import to in order for this command to work. Also you don't want to import any binaries or other files, so ensure you've done a new checkout or downloaded a fresh .tgz of their release before importing (i.e. ensure that what you're importing is fresh).

   ```
   cvs -d :ext:username@mao:/usr/local/cvsroot \
   import -kk users/sverret/testProject vendorName vendorRelease_v1_0
   ```

2. After the initial import we will tag the version as our own. To do this we need to use a release number that we understand. In the example below we use the release tag drdcRelease_v1_0

   ```
   cvs -d :ext:username@mao:/usr/local/cvsroot \
   rtag -r vendorRelease_v1_0 drdcRelease_v1_0 users/sverret/testProject
   ```

3. Finally check out the repository HEAD:

   ```
   cvs -d :ext:username@mao:/usr/local/cvsroot \
   checkout -kk users/sverret/testProject
   ```

   It is important to realize that the directory created on your personal computer is just a Sandbox for you to experiment in and play with. Once you've created your "sand castle" and you feel everything is correct (i.e. you've compiled and tested your changes), then and only then should you commit your changes to the repository which resides on the CVS server mao.

   ** **NOTE** It is possible to check out the vendor's code by typing the command

   ```
   cvs -d :ext:username@mao:/usr/local/cvsroot \
   checkout -kk -r vendorRelease_v1_0 users/sverret/testProject
   ```

but a user will NOT be able to commit files from this checkout, because it is NOT a branch but a sticky tag.

The normal procedure for importing code is below. This procedure assumes that the current ULM release in the repository is ulm_v1_4. Once the new version of code from ULM is imported it will take on the version number ulm_v1_5. Similarly this procedure assumes that the current version of DRDC's code is drdc_v1_4. On the way to drdc_v1_5 we will use the release number drdc_v1_4_1 to signify a version between drdc_v1_4 and drdc_v1_5.

1. Locate your ULM Miro code directory that contains ULMs code **FROM** ULM. In this example it will be called miroFromUlm. Next update your code with the latest updates.

   ```
   cd /PATH/miroFromUlm
   cvs update -Pd
   ```

2. If there were changes to the ULM Miro code from ULM then before we import anything into the ulmMiro repository we first need to label it.

   ```
   cd /PATH/ulmMiro
   cvs tag -c drdc_v1_4_1
   ```

   OR

   ```
   cvs -d :ext:username@mao:/usr/local/cvsroot \
   rtag -r HEAD drdc_v1_4_1 projects/cohort/miro
   ```

   which automatically tags the HEAD with the label *drdc_v1_4_1*. The difference between the two methods is the *-c* option in the first method that first checks if any files need to be committed before tagging the HEAD, whereas the second option can be run from anywhere and thus tags the HEAD automatically.

3. Import the code into the Defence R&D Canada – Suffield CVS ulmMiro repository using appropriate vendor tags and release tags.

   ```
   cvs -d :ext:username@mao:/usr/local/cvsroot import -kk \
   projects/cohort/miro ulm ulm_v1_5
   ```

   Replace *username* with an appropriate name, e.g. sverret. The vendor tag in this instance is *ulm* while the release tag is *ulm_v1_5*. Ensure that the release tag you use **HAS NOT BEEN USED BEFORE** and increments from the previous tag.

4. The results of the previous command will most likely yield some sort of conflict. You will probably get an output similar to this:

   ```
   C projects/cohort/miro/file1.txt
   U projects/cohort/miro/.cvsignore

   1 conflicts created by this import.
   ```

   Use the following command to help the merge:

   ```
   cvs -d :ext:sverret@mao:/usr/local/cvsroot checkout \
   -j<prev_rel_tag> -julm_v1_5 projects/cohort/miro
   ```

In this text they will give you an example command to try and workout the conflicts.

5. To work out conflicts you need to checkout another sandbox which merges the changes between two different versions. This exact command is given below with commentary.

   ```
   cvs -d :ext:sverret@mao:/usr/local/cvsroot checkout \
   -jdrdc_v1_4_1 -julm_v1_5 projects/cohort/miro
   ```

   This command will merge the differences between the files labeled *drdc_v1_4_1* and *ulm_v1_5* and checkout a new sandbox containing the merged results. If you're lucky you'll get a result like this:

   ```
   cvs checkout: Updating projects/cohort/miro
   U projects/cohort/miro/file1.txt
   RCS file: /usr/local/cvsroot/projects/cohort/miro/file1.txt,v
   retrieving revision 1.1
   retrieving revision 1.1.1.6
   Merging differences between 1.1 and 1.1.1.6 into file1.txt
   projects/cohort/miro/file1.txt already contains \
   the differences between 1.1 and 1.1.1.6
   ```

   If not, you might see conflicts which require fixing. In this case you will have to find the file, review what has been merged, and make the corrections.

   A script that looks for the words "already contains" or "rcsmerge" or "conflict" or "Merging" to see if there were conflicts would be nice to tell you which files had the conflicts.

   ** **NOTE** The numbers 1.1 and 1.1.1.6 have nothing to do with the numbers we used to tag various releases, e.g. drdc_v1_4_1.

6. Once all conflicts are fixed, and you must make and test your changes by first of all making

   ```
   ./bootstrap
   ./configure
   make
   ```

   and then testing by compiling drdcMiro. Finally, you must commit the files you changed. You can do this by typing in the root directory of the sandbox:

   ```
   cvs commit
   ```

   After committing all of the changes it'd be useful to now tag the HEAD as a new drdc release.

   ```
   cvs tag -c drdc_v1_5
   ```

   It's nice to have the DRDC release number correspond with the ULM release number we used at the start.

A situation may also arise such that we would like to change ULM's Miro code and submit it back to ULM for adoption. Below lists the procedure to do this:

1. First you need to make a patch file and you need to follow the following steps.

```
cvs tag -c drdcPatchTo_v1_5
```

The above command labels the HEAD of the ulmMiro repository with the label drdcPatchTo_v1_5.

2. Next we want to create the patch file.

```
cvs -d :ext:sverret@mao:/usr/local/cvsroot rdiff \
-r ulm_v1_5 -r drdcPatchTo_v1_5 projects/cohort/miro > v1_5.patch
```

This creates the file v1_5.patch which can be mailed to ULM along with a request to patch their repository.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada – Suffield
PO Box 4000, Station Main, Medicine Hat, AB,
Canada T1A 8K6

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

AISS Miro Manual: A Rough Guide (U)

4. AUTHORS (last name, first name, middle initial)

Giesbrecht, J.; Collier, J.; Broten, G.; Verret, S.; Monckton, S.

5. DATE OF PUBLICATION (month and year of publication of document)	6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc).	6b. NO. OF REFS (total cited in document)
December 2006	66	6

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

Technical Memorandum

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address).

9a. PROJECT NO. (the applicable research and development project number under which the document was written. Specify whether project).	9b. GRANT OR CONTRACT NO. (if appropriate, the applicable number under which the document was written).

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.)	10b. OTHER DOCUMENT NOs. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Suffield TM 2006-115	

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Defence departments and defence contractors; further distribution only as approved
( ) Defence departments and Canadian defence contractors; further distribution only as approved
( ) Government departments and agencies; further distribution only as approved
( ) Defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected).

Unlimited

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

The Autonomous Land Systems (ALS) and Cohort projects undertaken by the Autonomous Intelligent System Section (AISS) at Defence R&D Canada – Suffield research and develop autonomous unmanned vehicles. A software architecture provides a common software framework allowing researchers to easily implement, test, and share their research or algorithms. After a thorough review of existing robotics toolkits, the “Miro” framework was chosen as a basis on which to build DRDC’s Architechture for Autonomy. This document gives the reader background on “Miro” and the detailed procedure used to install the toolkit and all associated applications. Second, the basic concepts of “Miro” are described including the Interface Definition Language, Naming Service, Polled Mode Data Transfer, Event Driven Data Transfer and the Miro Directory Structure. Third, various “Miro” utilities and several “Miro” examples are described. Finally, more detailed “Miro” examples including how to add services to the Automake structure, the OMG Interface Definition Language and Parameter Files are discussed.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Miro, multi-robot systems, autonomous robots, CORBA